NumPy Functions Explained

#1: Numpy Array Creation Methods

- *Create a NumPy array from a python list*:
  You can create a numpy array from a given list using the np.array() method as shown below:

```
a = [1, 2, 3]
np.array(a)
```

- To check the datatype of the array, we can use the type method available in Python:

```
a = [1, 2, 3]
type(np.array(a))
```

- You can also create a multi-dimensional numpy array using a list of lists with the np.array() method:

```
a = [[1,2,3], [4,5,6]]
np.array(a)
```

- To specify the datatype for the numpy array, pass the dtype argument in the np.array() method as shown below:

```
a = [[1,2,3], [4,5,6]]
np.array(a, dtype = np.float32)
```

- Create a NumPy array of Zeros
  If you want to create a numpy array of a particular shape and all entries filled with zeros, you can do so using the np.zeros() method in numpy as follows:

```
np.zeros(5) #for 1-D arrays

np.zeros((2, 3)) #For multi-dimensional numpy arrays
```

- Create a NumPy array of Ones

  Similar to the above method, to create a numpy array filled with ones, you can use the np.ones() method as demonstrated below:

```
np.ones((2, 3))
```

- Create an Identity Numpy Array

  An identity matrix is a matrix with diagonal elements as “1" and all other elements as “0”. The np.eye() method returns a 2-D array with ones on the diagonal and zeros elsewhere.

```
np.eye(3)
```

- Create an equally spaced Numpy Array with a specific step

  If you want to generate equally spaced values within a given interval, you can do so using the np.arange() method:

```
np.arange(5, 11, 2)  #Generate values start=5 to step=11 with step=2.
```

- Create an equally spaced Numpy Array with a specific array size

  This is similar to np.arange() discussed above but with np.linspace(), you can generate num numbers in an interval that are evenly spaced.

```
np.linspace(start = 10, stop = 20, num = 5)
```

- Generate a random numpy array

  To generate a random array of integers, use the np.random.randint() method:

```
np.random.randint(low = 5, high = 16, size = 5)
```

- To generate random float samples, however, use the np.random.random() method:

```
np.random.random(size = 10)
```

- Generate NumPy Array from a Pandas Series
  If you want to convert a Pandas' series to a numpy array, you can do that using either np.array() or np.asarray() method:

```
s = pd.Series([1,2,3,4], name = "col")
np.array(s)
```

#2 NumPy Array Manipulation Methods

Next, we'll discuss some of the most widely-used methods to manipulate a numpy array.

1: Shape of the NumPy Array
You can determine the shape of the NumPy array using the np.shape() method of the ndarray.shape attribute of the NumPy array as follows:

```
a = np.ones((2, 3))
print("Shape of the array - Method 1:", np.shape(a))
print("Shape of the array - Method 2:", a.shape)
```

2: Reshape the NumPy Array
Reshaping refers to giving a new shape to your NumPy array without changing its data. You can alter the shape using the np.reshape() method:

```
a = np.arange(10)
a.reshape((2, 5))
```

3: Transpose the NumPy Array

If you want to transpose a numpy array, you can do so using the np.transpose() method or ndarray.T as demonstrated below:

```
a = np.arange(12).reshape((6, 2))
a.transpose()
```

4: Concatenate multiple NumPy arrays to form one NumPy Array

You can use the np.concatenate() method to join a sequence of arrays and obtain a new numpy array:

```
## Concatenate Row-wise

a = np.array([[1, 2], [3, 4]])
b = np.array([[5, 6]])
np.concatenate((a, b), axis=0)

## Concatenate Column-wise

a = np.array([[1, 2], [3, 4]])
b = np.array([[5, 6]])
np.concatenate((a, b.T), axis=1)


## Concatenate to generate a flat NumPy Array

a = np.array([[1, 2], [3, 4]])
b = np.array([[5, 6]])
np.concatenate((a, b), axis=None)
```

Note:
axis=0 is same as np.vstack().
axis=1 is same as np.hstack().

5: Flatten NumPy Array

If you want to collapse the entire NumPy array into a single dimension, you can use the ndarray.flatten() method as shown below:

```
a = np.array([[1,2], [3,4]])
a.flatten()
```

6: Unique Elements of a NumPy Array

To determine the unique elements of a numpy array, use the np.unique() method as demonstrated below:

```
a = np.array([[1, 2], [2, 3]])
np.unique(a)

## Return Unique Rows

a = np.array([[1, 2, 3], [1, 2, 3], [2, 3, 4]])
np.unique(a, axis=0)


## Return Unique Columns

a = np.array([[1, 1, 3], [1, 1, 3], [1, 1, 4]])
np.unique(a, axis=1)
```

7: Squeeze a NumPy Array

If you want to remove axes of length one from your numpy array, use the np.squeeze() method. This is illustrated below:

```
x = np.array([[[0], [1], [2]]])
x.shape
np.squeeze(x)
X.shape
```

8: Transform NumPy Array to Python List

To obtain a python list from a NumPy array, use the ndarry.tolist() method as shown below:

```
a = np.array([[1, 1, 3], [1, 1, 3], [1, 1, 4]])
a.tolist()
```

Horizontal Stacking
hstack will append one array at the end of another.

```
a = np.array([1,2,3,4,5])
b = np.array([1,4,9,16,25])
np.hstack((a,b))
```

Vertical Stacking
vstack will stack one array on top of another.

```
np.vstack((a,b))
```

#3 Mathematical Operations on NumPy Arrays

NumPy offers a wide variety of element-wise mathematical functions you can apply to NumPy arrays. You can read all the mathematical operations available in numpy documentation. Below, let's discuss some of the most commonly used ones.

1:Trigonometric Functions

```
a = np.array([1,2,3])
print("Trigonometric Sine   :", np.sin(a))
print("Trigonometric Cosine :", np.cos(a))
print("Trigonometric Tangent:", np.tan(a))
```

2: Rounding Functions
Return the element-wise floor using the np.floor() method.
Return the element-wise ceiling using the np.ceil() method.
Round to the nearest integer using the np.rint() method.

```
a = np.linspace(1, 2, 5)
```

```
array([1.  , 1.25, 1.5 , 1.75, 2.  ])

np.floor(a)
array([1., 1., 1., 1., 2.])

np.ceil(a)
array([1., 2., 2., 2., 2.])

np.rint(a)
array([1., 1., 2., 2., 2.])
```

Round to given number of decimals using the np.round_() method:

```
a = np.linspace(1, 2, 7)
np.round_(a, 2) # 2 decimal places
```

4: Exponents and logarithms
Calculate the element-wise exponential using the np.exp() method.
Calculate the element-wise natural logarithm using the np.log() method.

5: Sum and Product
Use the np.sum() method to calculate the sum of array elements:

```
a = np.array([[1, 2], [3, 4]])
np.sum(a)

np.sum(a, axis = 0)
array([4, 6])
np.sum(a, axis = 1)
array([3, 7])
```

Use the np.prod() method to calculate the product of array elements:

```
a = np.array([[1, 2], [3, 4]])
```

```
>>> np.prod(a)
24

>>> np.prod(a, axis = 0)
array([3, 8])

>>> np.sum(a, axis = 1)
array([2, 12])
```

6: Square Root

Use the np.sqrt() method to compute the square root of array elements:

```
a = np.array([[1, 2], [3, 4]])
np.sqrt(a)
```

#4 Matrix and Vector Operations

1:Dot Product

If you want to calculate the dot product of two NumPy arrays, use the np.dot() method:

```
      a = np.array([[1, 2], [3, 4]])
      b = np.array([[1, 1], [1, 1]])
np.dot(a, b)
```

2: Matrix Product

To calculate the matrix product of two NumPy arrays, use the np.matmul() or the @ operator in Python:

```
a = np.array([[1, 2], [3, 4]])
b = np.array([[1, 1], [1, 1]])

>>> np.matmul(a, b)
array([[3, 3],
       [7, 7]])

>>> a@b
array([[3, 3],
       [7, 7]])
```

3: Vector Norm

Vector norms represent a set of functions used to measure a vector's length.

- Use the np.linalg.norm() method to find the matrix or vector norm:

```
a = np.arange(-4, 5)

>>> np.linalg.norm(a) ## L2 Norm
7.745966692414834

>>> np.linalg.norm(a, 1) ## L1 Norm
20.0
```

4: Inner and Dot Product:

The inner product takes two vectors of equal size and returns a single number (scalar). This is calculated by multiplying the corresponding elements in each vector and adding up all of those products. In numpy, vectors are defined as one-dimensional numpy arrays.

To get the inner product, we can use either np.inner() or np.dot(). Both give the same results. The inputs for these functions are two vectors and they should be the same size.

Dot product:

The dot product is defined for matrices. It is the sum of the products of the corresponding elements in the two matrices. To get the dot product, the number of columns in the first matrix should be equal to the number of rows in the second matrix.
There are two ways to create matrices in numpy. The most common one is to use the numpy ndarray class. Here we create two-dimensional numpy arrays (ndarray objects). The other one is to use the numpy matrix class. Here we create matrix objects.

The dot product of both ndarray and matrix objects can be obtained using *np.dot()*.

```
import numpy as np

# Vectors as numpy arrays
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])
print("a= ", a)
print("b= ", b)
print("\n inner prod:", np.inner(a, b))
print("dot:", np.dot(a, b))


# Matrices as matrix objects
c = np.matrix([[1, 2], [3, 4]])
d = np.matrix([[5, 6, 7], [8, 9, 10]])
print("\ndot product of two matrix objects")
print(np.dot(c, d))


## elementwise mult of arrays using * operator
print("\n* operation on two ndarray objects (Elementwise)")
print(a * b)
print("\n* operation on two matrix objects (same as np.dot())")
print(c * d)
```

Note that when multiplying two ndarray objects using the * operator, the result is the element-by-element multiplication. On the other hand, when multiplying two matrix objects using the * operator, the result is the dot (matrix) product which is equivalent to the np.dot() as previous.

5: Transpose

The transpose of a matrix is found by switching its rows with its columns. We can use np.transpose() function or NumPy ndarray.transpose() method or ndarray.T (a special method which does not require parentheses) to get the transpose. All give the same output.

```
import numpy as np

a = np.array([[1, 2], [3, 4], [5, 6]])
print("a = ")
print(a)

print("\nWith np.transpose(a) function")
print(np.transpose(a))

print("\nWith ndarray.transpose() method")
print(a.transpose())

print("\nWith ndarray.T short form")
print(a.T)
```

The transpose can also be applied to a vector. However, technically, a one-dimensional numpy array cannot be transposed.

```
import numpy as np
a = np.array([1, 2, 3])
print("a = ")
print(a)
print("\na.T = ")
print(a.T)
```

If you really want to transpose a vector, it should be defined as a two-dimensional numpy array with double square brackets.

```
import numpy as np
a = np.array([[1, 2, 3]])
print("a = ")
print(a)
print("\na.T = ")
print(a.T)
```

6: Trace

The trace is the sum of diagonal elements in a square matrix. There are two methods to calculate the trace. We can simply use the trace() method of an ndarray object or get the diagonal elements first and then get the sum.

```
import numpy as np
a = np.array([[2, 2, 1],
              [1, 3, 1],
              [1, 2, 2]])
print("a = ")
print(a)
print("\nTrace:", a.trace())
print("Trace:", sum(a.diagonal()))
```

7: Rank

The rank of a matrix is the dimensions of the vector space spanned (generated) by its columns or rows. In other words, it can be defined as the maximum number of linearly independent column vectors or row vectors.

The rank of a matrix can be found using the matrix_rank() function which comes from the numpy linalg package.

```
import numpy as np
a = np.arange(1, 10)
a.shape = (3, 3)
print("a = ")
print(a)
rank = np.linalg.matrix_rank(a)
print("\nRank:", rank)
```

8: Determinant

The determinant of a square matrix can be calculated det() function which also comes from the numpy linalg package. If the determinant is 0, that matrix is not invertible. It is known as a singular matrix in algebra terms.

```
import numpy as np
a = np.array([[2, 2, 1],
              [1, 3, 1],
              [1, 2, 2]])
print("a = ")
print(a)
det = np.linalg.det(a)
print("\nDeterminant:", np.round(det))
```

9: True inverse

The true inverse of a square matrix can be found using the inv() function of the numpy linalg package. If the determinant of a square matrix is not 0, it has a true inverse.

```
import numpy as np
a = np.array([[2, 2, 1],
              [1, 3, 1],
              [1, 2, 2]])
print("a = ")
print(a)
det = np.linalg.det(a)
print("\nDeterminant:", np.round(det))
inv = np.linalg.inv(a)
print("\nInverse of a = ")
print(inv)
```

If you try to compute the true inverse of a singular matrix (a square matrix whose determinant is 0), you will get an error.

```
import numpy as np
a = np.array([[2, 8],
              [1, 4]])
print("a = ")
print(a)
det = np.linalg.det(a)
print("\nDeterminant:", np.round(det))
inv = np.linalg.inv(a)
print("\nInverse of a = ")
print(inv)
```

10: Pseudo inverse

The pseudo (not genuine) inverse can be calculated even for a singular matrix (a square matrix whose determinant is 0) using the pinv() function of the numpy linalg package.

```
import numpy as np
a = np.array([[2, 8],
              [1, 4]])
print("a = ")
print(a)
det = np.linalg.det(a)
print("\nDeterminant:", np.round(det))
pinv = np.linalg.pinv(a)
print("\nPseudo Inverse of a = ")
print(pinv)
```

There is no difference between true inverse and pseudo-inverse if a square matrix is non-singular (determinant is not 0).

11: Flatten

Flatten is a simple method to transform a matrix into a one-dimensional numpy array. For this, we can use the flatten() method of an ndarray object.

```
import numpy as np
a = np.arange(1, 10)
a.shape = (3, 3)
print("a = ")
print(a)
print("\nAfter flattening")
print("------------------")
print(a.flatten())
```

12: Eigenvalues and eigenvectors

Let A be an n x n matrix. A scalar λ is called an eigenvalue of A if there is a non-zero vector x satisfying the following equation.

$$Ax = \lambda x$$

The vector x is called the eigenvector of A corresponding to λ.

In numpy, both eigenvalues and eigenvectors can be calculated simultaneously using the eig() function.

```
import numpy as np
a = np.array([[2, 2, 1],
              [1, 3, 1],
              [1, 2, 2]])
print("a = ")
print(a)
w, v = np.linalg.eig(a)
print("\nEigenvalues:")
print(w)
print("\nEigenvectors:")
print(v)
```

The sum of eigenvalues (1+5+1=7) is equal to the trace (2+3+2=7) of the same matrix! The product of the eigenvalues (1x5x1=5) is equal to the determinant (5) of the same matrix!

Eigenvalues and eigenvectors are extremely useful in the Principal Component Analysis (PCA). In PCA, the eigenvectors of the correlation or covariance matrix represent the principal components (the directions of maximum variance) and the corresponding eigenvalues represent the amount of variation explained by each principal component.

#5 Sorting Methods

1: Sort a NumPy Array
To sort the array in-place, use the ndarray.sort() method.

```
a = np.array([[1,4],[3,1]])

>>> np.sort(a) ## sort based on rows
array([[1, 4],
       [1, 3]])

>>> np.sort(a, axis=None) ## sort the flattened array
array([1, 1, 3, 4])

>>> np.sort(a, axis=0) ## sort based on columns
array([[1, 1],
       [3, 4]])
```

2: Order of Indices in a Sorted NumPy Array
To return the order of the indices that would sort the array, use the np.argsort() method:

```
x = np.array([3, 1, 2])
np.argsort(x)
```

#6 Searching Methods

1: Indices corresponding to Maximum Values
To return the indices of the maximum values along an axis, use the np.argmax() method as demonstrated below:

```
>>> a = np.random.randint(1, 20, 10).reshape(2,5)
array([[15, 13, 10,  1, 18],
       [14, 19, 19, 17,  8]])

>>> np.argmax(a) ## index in a flattened array
6

>>> np.argmax(a, axis=0) ## indices along columns
array([0, 1, 1, 1, 0])

>>> np.argmax(a, axis=1) ## indices along rows
array([4, 1])
```

To find the index in a non-flattened array, you can do the following:

```
ind = np.unravel_index(np.argmax(a), a.shape)
Ind
```

2: Indices corresponding to Minimum Values
Similarly, if you want to return the indices of the minimum values along an axis, use the np.argmin() method as demonstrated below:

```
>>> a = np.random.randint(1, 20, 10).reshape(2,5)
```

```
array([[15, 13, 10,  1, 18],
       [14, 19, 19, 17,  8]])

>>> np.argmin(a) ## index in a flattend array
3

>>> np.argmin(a, axis=0) ## indices along columns
array([1, 0, 0, 0, 1])

>>> np.argmin(a, axis=1) ## indices along rows
array([3, 4])
```

3: Search based on condition

If you want to select between two arrays based on a condition, use the np.where() method as shown below:

```
>>> a = np.random.randint(-10, 10, 10)
array([ 2, -3,  6, -3, -8,  4, -6, -2,  6, -4])

>>> np.where(a < 0, 0, a)
array([2, 0, 6, 0, 0, 4, 0, 0, 6, 0])
"""
if element < 0:
    return 0
else:
    return element
"""
```

4: Indices of Non-Zero Elements

To determine the indices of non-zero elements in a NumPy Array, use the np.nonzero() method:

```
a = np.array([[3, 0, 0], [0, 4, 0], [5, 6, 0]])
np.nonzero(a)
```

#7 Statistical Methods

Next, let's look at the methods to compute standard statistics on NumPy Arrays. You can find all the statistical techniques supported by NumPy here.

1:Mean
Mean is the sum of the elements divided by its sum and given by the following formula:

$$\bar{x} = \frac{1}{n}(x_1 + x_2 + \ldots + x_n)$$

It calculates the mean by adding all the items of the arrays and then divides it by the number of elements. We can also mention the axis along which the mean can be calculated.

To find the mean of the values in a numpy array along an axis, use the np.mean() method as demonstrated below:

```
a = np.array([[1, 2], [3, 4]])

>>> np.mean(a)
2.5

>>> np.mean(a, axis = 1) ## along the row axis
array([1.5, 3.5])

>>> np.mean(a, axis = 0) ## along the column axis
array([2., 3.])
```

2: Median

Median is the middle element of the array. The formula differs for odd and even sets.

Odd	Even
$\frac{n+1}{2}$	$\frac{n}{2}, \frac{n}{2}+1$

It can calculate the median for both one-dimensional and multi-dimensional arrays. Median separates the higher and lower range of data values.

To compute the median of a numpy array, use the np.median() method.

```
a = np.array([[1, 2], [3, 4]])

>>> np.median(a)
2.5

>>> np.median(a, axis = 1) ## along the row axis
array([1.5, 3.5])

>>> np.median(a, axis = 0) ## along the column axis
array([2., 3.])
```

3: Standard Deviation

Standard deviation is the square root of the average of square deviations from mean. The formula for standard deviation is:

$$\mathrm{SD} = \sqrt{\frac{\sum |x - \bar{x}|^2}{n}}$$

To compute the standard deviation of a numpy array along a specified array, use the np.std() method.

```
a = np.array([[1, 2], [3, 4]])

>>> np.std(a)
1.118033988749895

>>> np.std(a, axis = 1) ## along the row axis
array([0.5, 0.5])

>>> np.std(a, axis = 0) ## along the column axis
array([1., 1.])
```

4: np.amin()- This function determines the minimum value of the element along a specified axis.

```
import numpy as np
arr= np.array([[1,23,78],[98,60,75],[79,25,48]])
print(arr)
#Minimum Function
print(np.amin(arr))
```

5: np.amax()- This function determines the maximum value of the element along a specified axis.

```
#Maximum Function
print(np.amax(arr))
```

6: np.var – It determines the variance.

Variance is the average of the square deviations. Following is the formula for the same:

$$\sigma^2 = \frac{\sum (x_r - \overline{x})^2}{n}$$

```
import numpy as np

a = np.array([5,6,7])

print(a)

print(np.var(a))
```

7: np.ptp()- It returns a range of values along an axis.

```
NumPy np.ptp() function is useful to determine the range of values along an
axis.

a = np.array([[2,10,20],[6,10,60]])
print(np.ptp(a,0))
```

8: np.average()- It determines the weighted average

NumPy np.average() function determines the weighted average along with the multi-dimensional arrays. The weighted average is calculated by multiplying the component by

its weight, the weights are specified separately. If weights are not specified it produces the same output as mean.

```
import numpy as np
a = np.array([5,6,7])
print(a)
#without weight same as mean
print(np.average(a))
#with weight gives weighted average
wt = np.array([8,2,3])
print(np.average(a, weights=wt))
```

9: np.percentile()- It determines the nth percentile of data along the specified axis.

Calculates the percentile of elements in an array.

It has the following syntax:
numpy.percentile(input, q, axis)

The accepted parameters are:

- input: it is the input array.
- q: it is the percentile which it calculates of the array elements between 0-100.
- axis: it specifies the axis along which calculation is performed.

```
a = np.array([2,10,20])
print(a)
print(np.percentile(a,10,0))
```